3. 메타프로그래밍

3.1 메타프로그래밍_I

3.2 메타프로그래밍_II

3.3 메타프로그래밍_III

3.4 메타프로그래밍_IV

함수 감싸기


```
import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

"함수에 추가적인 처리(로깅,타이밍 등)를 하는 래퍼 레이어를 넣고 싶다."

함수 감싸기

```
if __name__ == '__main__':
    @timethis
    def countdown(n:int):
        while n > 0:
            n -= 1

    countdown(100000)
    countdown(10000000)
```

```
countdown 0.0048801898956298 83
countdown 0.4635972976684570 3
```

"함수에 추가적인 코드를 감싸려면 데코레이터 함수를 정의한다."

함수 메타데이터 보존


```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

"데코레이터를 작성했다. 하지만 이를 함수에 적용할 때 이름, 독 문자열, 주석, 호출 시그니처와 같은 주요 메타데이터가 사라진다."

함수 메타데이터 보존

```
if __name__ == '__main__':
    @timethis
    def countdown(n:int):
        '''
        Counts down
        '''
        while n > 0:
            n -= 1

    countdown(100000)
    print('Name:', countdown.__name__)
    print('Docstring:', repr(countdown.__doc__))
    print('Annotations:', countdown.__annotations__)
```

```
countdown 0.004880428314208984
Name: countdown
Docstring: '\n        Counts down\n        '
Annotations: {'n': <class 'int'>}
```

"데코레이터를 작성할 때는 언제나 functools 라이브러리의 @wraps 데코레이터를 래퍼 함수에 적용해야 한다."

데코레이터 풀기


```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper
```

"함수에 데코레이터를 적용했는데, 이를 '취소'하고 원본 함수에 접근하고 싶다."

데코레이터 풀기

```
@decorator1
@decorator2
def add(x, y):
    return x + y

print(add(2,3))

print(add.__wrapped__(2,3))
```

```
Decorator 1
Decorator 2
5
Decorator 2
5
```

"@wraps를 사용해서 데코레이터를 올바르게 구현했다고 가정하면, 원본 함수에는 __wrapped__ 속성으로 접근할 수 있다."

매개변수를 받는 데코레이터 정의


```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

"매개변수를 받는 데코레이터를 작성하고 싶다."

매개변수를 받는 데코레이터 정의


```
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')

if __name__ == '__main__':
    import logging
    logging.basicConfig(level=logging.DEBUG)
    print(add(2,3))
    spam()
```

```
DEBUG:__main__:add
CRITICAL:example:spam
```

```
5
Spam!
```

"가장 외부에 있는 logged() 함수는 매개 변수 인자를 받고 이를 가지고 내부 데코레이터 함수에서 사용하게 된다."

사용자가 조절 가능한 속성을 가진 데코레이터

```
from functools import wraps, partial
import logging

def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__
```

"함수를 감싸는 데코레이터 함수를 작성하는데, 사용자가 실행 시간에 동작성을 변경할 수 있도록 속성을 조절하도록 하고 싶다."

사용자가 조절 가능한 속성을 가진 데코레이터

```
        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper
    return decorate
```

"nonlocal 변수 선언을 통해 내부 변수를 바꾸는 접근자 함수를 만들어 보자."

사용자가 조절 가능한 속성을 가진 데코레이터

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1


@logged(logging.DEBUG)
@timethis
def countdown2(n):
    while n > 0:
        n -= 1
```

사용자가 조절 가능한 속성을 가진 데코레이터

```
if __name__ == '__main__':
    import logging
    logging.basicConfig(level=logging.DEBUG)
    print(add(2, 3))

    add.set_message('Add called')
    print(add(2, 3))

    add.set_level(logging.WARNING)
    print(add(2, 3))

    countdown(100000)
    countdown.set_level(logging.CRITICAL)
    countdown(100000)

    countdown2(100000)
    countdown2.set_level(logging.CRITICAL)
    countdown2(100000)
```

사용자가 조절 가능한 속성을 가진 데코레이터

```
DEBUG:__main__:add
DEBUG:__main__:Add called
WARNING:__main__:Add called
DEBUG:__main__:countdown
CRITICAL:__main__:countdown
DEBUG:__main__:countdown2
CRITICAL:__main__:countdown2
```

```
5
5
5
countdown 0.004878997802734375
countdown 0.0058553218841552734
countdown2 0.0049116611480712891
countdown2 0.0048792362213134766
```

"중요한 부분은 래퍼에 속성으로 붙이는 접근자 함수(set_message() 와 set_level() 등) 에 있다. 모든 접근자는 내부 파라미터를 nonlocal 할당을 사용하여 조절 하도록 한다."

옵션 매개변수를 받는 데코레이터

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__
    @wraps(func)
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)
    return wrapper
```

"@decorator와 같이 매개변수가 없거나 @decorator(x,y,z)와 같이 옵션 매개변수가 있는 데코레이터 하나를 작성하고 싶다."

옵션 매개변수를 받는 데코레이터


```
@logged
def add(x, y):
    return x + y

@logged()
def sub(x, y):
    return x - y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')

if __name__ == '__main__':
    import logging
    logging.basicConfig(level=logging.DEBUG)
    add(2,3)
    sub(2,3)
    spam()
```

옵션 매개변수를 받는 데코레이터

```
DEBUG:__main__:add
DEBUG:__main__:sub
CRITICAL:example:spam
```

```
Spam!
```

"데코레이터는 간단한 형식(즉, @logged)이나 매개변수를 받는 형식 (즉, @logged(level=logging.CRITICAL, name='example'))으로 사용 가능하다."

3. 메타프로그래밍

3.1 메타프로그래밍_I

3.2 메타프로그래밍_II

3.3 메타프로그래밍_III

3.4 메타프로그래밍_IV

데코레이터를 사용해서 함수에서 타입 확인 강제


```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        if not __debug__:
            return func

        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_types[name])
                            )
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

데코레이터를 사용해서 함수에서 타입 확인 강제


```
@typeassert(int, int)
def add(x, y):
    return x + y

@typeassert(int, z=int)
def spam(x, y, z=42):
    print(x, y, z)

if __name__ == '__main__':
    print(add(2,3))
    try:
        add(2, 'hello')
    except TypeError as e:
        print(e)

    spam(1, 2, 3)
    spam(1, 'hello', 3)
    try:
        spam(1, 'hello', 'world')
    except TypeError as e:
        print(e)
```

```
5
Argument y must be <class 'int'>
1 2 3
1 hello 3
Argument z must be <class 'int'>
```

"함수의 매개변수의 타입을 강제적으로 확인하는 기능을 구현하고 싶다."

"모든 매개변수에 대해서 타입을 명시하거나 혹은 부분만 지정할 수 있어 데코레이터가 거느 정도 유연하다는 점을 알 수 있다."

데코레이터를 클래스의 일부로 정의


```
from functools import wraps

class A:
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

"데코레이터를 클래스 정의 내부에 정의하고 다른 함수나 메소드에 적용하고 싶다."

데코레이터를 클래스의 일부로 정의


```
a = A()

@a.decorator1
def spam():
    pass

@A.decorator2
def grok():
    pass

spam()
grok()
```

```
Decorator 1
Decorator 2
```

"데코레이터를 클래스 내부에 정의하기는 어렵지 않지만, 어떤 데코레이터를 적용할지 순서를 먼저 정렬해야 한다."

데코레이터를 클래스의 일부로 정의

```
class Person:
    first_name = property()
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

p = Person()
p.first_name = 'Dave'
print(p.first_name)
```

```
Dave
```

"내장 데코레이터 @property는 실제로 getter(), setter(), delete() 메소드를 가진 클래스이고 데코레이터처럼 동작한다.

클래스 데코레이터 정의


```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

"함수를 데코레이터로 감싸고 싶지만, 그 결과는 호출 가능 인스턴스가 되도록 하고 싶다. 그리고 데코레이터를 클래스 정의 내부와 외부에서 모두 사용하고 싶다.

클래스 데코레이터 정의

```
class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)

if __name__ == '__main__':
    print(add(2,3))
    print(add(4,5))
    print('ncalls:', add.ncalls)

    s = Spam()
    s.bar(1)
    s.bar(2)
    s.bar(3)
    print('ncalls:', Spam.bar.ncalls)
```

```
5
9
ncalls: 2
<__main__.Spam object at 0x000001CB4FA47908> 1
<__main__.Spam object at 0x000001CB4FA47908> 2
<__main__.Spam object at 0x000001CB4FA47908> 3
ncalls: 3
```

"데코레이터를 인스턴스로 정의 하려면 __call__()과 __get__() 메소드를 반드시 구현해야 한다."

클래스 데코레이터 정의


```
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper
```

"클로저와 nonlocal 변수를 데코레이터를 고려해 보자."

클래스 데코레이터 정의

```
@profiled
def add(x, y):
    return x + y

class Spam:
    @profiled
    def bar(self, x):
        print(self, x)

if __name__ == '__main__':
    print(add(2,3))
    print(add(4,5))
    print('ncalls:', add.ncalls())

    s = Spam()
    s.bar(1)
    s.bar(2)
    s.bar(3)
    print('ncalls:', Spam.bar.ncalls())
```

```
5
9
ncalls: 2
<__main__.Spam object at 0x000001CB4FA47DC8> 1
<__main__.Spam object at 0x000001CB4FA47DC8> 2
<__main__.Spam object at 0x000001CB4FA47DC8> 3
ncalls: 3
```

클래스와 스태틱 메소드에 데코레이터 적용


```
import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return r
    return wrapper

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper
```

클래스와 스태틱 메소드에 데코레이터 적용

```
@profiled
def add(x, y):
    return x + y

class Spam:
    @profiled
    def bar(self, x):
        print(self, x)

@timethis
@profiled
def countdown(n):
    while n > 0:
        n -= 1
```

"클래스나 스태틱 메소드에 데코레이터를 적용하고 싶다."

```
if __name__ == '__main__':
    print(add(2,3))
    print(add(4,5))
    print('ncalls:', add.ncalls())

    s = Spam()
    s.bar(1)
    s.bar(2)
    s.bar(3)
    print('ncalls:', Spam.bar.ncalls())

    countdown(100000)
    countdown(10000000)
    print(countdown.ncalls())
```

"클래스나 스태틱 메소드에 데코레이터를 적용하기는 간단하지만, 데코레이터를 @classmethod 또는 @staticmethod 앞에 적용하도록 주의하자."

```
5
9
ncalls: 2
<__main__.Spam object at 0x000001CB4F916988> 1
<__main__.Spam object at 0x000001CB4F916988> 2
<__main__.Spam object at 0x000001CB4F916988> 3
ncalls: 3
countdown 0.0039348602294921875
countdown 0.46652650833129883
2
```

클래스와 스태틱 메소드에 데코레이터 적용


```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper
```

클래스와 스태틱 메소드에 데코레이터 적용


```
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

클래스와 스태틱 메소드에 데코레이터 적용

```
if __name__ == '__main__':
    s = Spam()
    s.instance_method(10000000)
    Spam.class_method(10000000)
    Spam.static_method(10000000)
```

```
<__main__.Spam object at 0x000001CB4F9F1848> 10000000
0.48311805725097656
<class '__main__.Spam'> 10000000
0.4626448154449463
10000000
0.5026047229766846
```

감싼 함수에 매개변수를 추가하는 데코레이터 작성


```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper

@optional_debug
def spam(a, b, c):
    print(a, b, c)

spam(1, 2, 3)
spam(1, 2, 3, debug=True)
```

```
1 2 3
Calling spam
1 2 3
```

"감싼 함수의 호출 시그니처에 매개변수를 추가하는 데코레이터를 작성하고 싶다. 하지만 추가한 매개변수는 기존 함수의 호출 방식과 함께 사용할 수 없다."

"키워드로만 넣을 수 있는 인자를 사용하면 호출 시그니처에 매개변수를 추가할 수 있다."

클래스 정의 패치에 데코레이터 사용


```
def log_getattribute(cls):
    orig_getattribute = cls.__getattribute__

    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    cls.__getattribute__ = new_getattribute
    return cls

@log_getattribute
class A:
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass

if __name__ == '__main__':
    a = A(42)
    print(a.x)
    a.spam()
```

```
getting: x
42
getting: spam
```

"상속이나 메타클래스를 사용하지 않고, 클래스 정의의 일부를 조사, 재작성해서 동작성을 변경하고 싶다."

"위 코드는 __getattribute__ 특별 메소드가 로깅을 수행하도록 하는 클래스 데코레이터이다."

3. 메타프로그래밍

3.1 메타프로그래밍_I

3.2 메타프로그래밍_II

3.3 메타프로그래밍_III

3.4 메타프로그래밍_IV

인스턴스 생성 조절에 메타클래스 사용


```
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')

if __name__ == '__main__':
    try:
        s = Spam()
    except TypeError as e:
        print(e)

    Spam.grok(42)
```

```
Can't instantiate directly
Spam.grok
```

"싱글톤(singleton), 캐싱 등 기능 구현을 위해 인스턴스가 생성되는 방법을 변경하고 싶다."

인스턴스 생성 조절에 메타클래스 사용

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

인스턴스 생성 조절에 메타클래스 사용

```
if __name__ == '__main__':
    a = Spam()
    b = Spam()
    print(a is b)
```

```
Creating Spam
True
```

"오직 하나의 인스턴스만 생성된다."

인스턴스 생성 조절에 메타클래스 사용

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name
```

인스턴스 생성 조절에 메타클래스 사용

```
if __name__ == '__main__':
    a = Spam('foo')
    b = Spam('bar')
    print('a is b:', a is b)
    c = Spam('foo')
    print('a is c:', a is c)
```

```
Creating Spam('foo')
Creating Spam('bar')
a is b: False
a is c: True
```

"캐시 인스턴스를 구현한다."

클래스 속성 정의 순서 수집


```
from collections import OrderedDict

class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = name

    def __set__(self, instance, value):
        if not isinstance(value, self._expected_type):
            raise TypeError('Expected ' +str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str
```

클래스 속성 정의 순서 수집


```
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()

class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)
```

클래스 속성 정의 순서 수집


```
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

if __name__ == '__main__':
    s = Stock('GOOG',100,490.1)
    print(s.name)
    print(s.as_csv())
    try:
        t = Stock('AAPL','a lot', 610.23)
    except TypeError as e:
        print(e)
```

클래스 속성 정의 순서 수집

```
GOOG
GOOG,100,490.1
Expected <class 'int'>
```

"클래스 내부에 정의되는 속성과 메소드의 순서를 자동으로 기록해서 여러 목적에 사용하고 싶다."

"메타클래스를 시용하면 클래스 정의에 대한 정보를 쉽게 얻을 수 있다."

옵션 매개변수를 받는 메타클래스 정의

```
class MyMeta(type):
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        return super().__prepare__(name, bases)

    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        return super().__new__(cls, name, bases, ns)

    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        super().__init__(name, bases, ns)

class A(metaclass=MyMeta, debug=True, synchronize=True):
    pass

class B(metaclass=MyMeta):
    pass

class C(metaclass=MyMeta, synchronize=True):
    pass
```

"메타클래스에서 키워드 매개변수를 지원하려면, 다음과 같이 __prepare__(), __new__(), __init__() 메소드에 키워드로만 받는 인자를 사용해서 정의해야 한다."

*args, **kwargs에 매개변수 시그니처 강제


```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

*args, **kwargs에 매개변수 시그니처 강제

```
if __name__ == '__main__':
    s1 = Stock('ACME', 100, 490.1)
    print(s1.name, s1.shares, s1.price)

    s2 = Stock(shares=100, name='ACME', price=490.1)
    print(s2.name, s2.shares, s2.price)

    try:
        s3 = Stock('ACME', 100)
    except TypeError as e:
        print(e)

    try:
        s4 = Stock('ACME', 100, 490.1, '6/1/2020')
    except TypeError as e:
        print(e)

    try:
        s5 = Stock('ACME', 100, name='ACME', price=490.1)
    except TypeError as e:
        print(e)
```

*args, **kwargs에 매개변수 시그니처 강제

```
ACME 100 490.1
ACME 100 490.1
missing a required argument: 'price'
too many positional arguments
multiple values for argument 'name'
```

*"제너럴한 목적은 *args와 ** kwargs를 사용하는 함수나 메소드를 작성했지만, 전달 받은 매개변수가 특정 함수 호출 시그니처에 일치하는지 확인 하고 싶다."*

"함수 호출 시그니처를 관리하는 모든 문제에는 inspect 모듈에 있는 시그니처 기능을 사용해야 한다."

*args, **kwargs에 매개변수 시그니처 강제

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields',[]))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)
```

*args, **kwargs에 매개변수 시그니처 강제


```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields',[]))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']
```

*args, **kwargs에 매개변수 시그니처 강제

```
if __name__ == '__main__':
    s1 = Stock('ACME', 100, 490.1)
    print(s1.name, s1.shares, s1.price)

    s2 = Stock(shares=100, name='ACME', price=490.1)
    print(s2.name, s2.shares, s2.price)

    try:
        s3 = Stock('ACME', 100)
    except TypeError as e:
        print(e)

    try:
        s4 = Stock('ACME', 100, 490.1, '6/1/2020')
    except TypeError as e:
        print(e)

    try:
        s5 = Stock('ACME', 100, name='ACME', price=490.1)
    except TypeError as e:
        print(e)
```

*args, **kwargs에 매개변수 시그니처 강제

```
ACME 100 490.1
ACME 100 490.1
missing a required argument: 'price'
too many positional arguments
multiple values for argument 'name'
```

"스스로 매개변수를 확인하려할 때 코드가 지저분 해진다는 단점이 있다 ."

"커스텀 시그니처를 정의할 때, 시그니처를 특별 속성 __signature__에 저장하는 것이 유용하다. 이렇게 하면, 코드 조사를 위해 inspect 모듈을 사용하는 코드가 시그니처를 보고 호출 규칙으로 보고한다."

클래스 코딩 규칙 강제


```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self):
        pass

print('**** About to generate a TypeError')
class B(Root):
    def fooBar(self):
        pass
```

```
**** About to generate a TypeError
TypeError: Bad attribute name: fooBar
```

"메타클래스로 클래스 정의를 모니터링할 수 있다. 기본적인 메타클래스는 일반적으로 type을 상속 받아 정의하고 __new__() 메소드를 재정의 한다."

클래스 코딩 규칙 강제


```
from inspect import signature
import logging

class MatchSignaturesMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            prev_dfn = getattr(sup,name,None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                    value.__qualname__, str(prev_sig), str(val_sig))
```

클래스 코딩 규칙 강제

```
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

class B(A):
    def foo(self, a, b):
        pass

    def spam(self,x,z):
        pass
```

```
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)
WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
```

"키워드 매개변수에 의존해서 메소드에 전달하는 코드는 서브클래스에서 매개변수 이름을 바꾸었을때 문제가 발생한다."

클래스를 프로그램적으로 정의

```
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price

def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}
```

"새로운 클래스 객체를 생성하는 코드를 작성 중이다. 클래스 소스 코드를 문자열 형식으로 만들고 exec()와 같은 함수로 실행할까 생각 중인데, 조금 더 보기 좋은 해결책을 찾고 있다."

클래스를 프로그램적으로 정의

```
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))

if __name__ == '__main__':
    s = Stock('ACME', 50, 91.1)
    print(s)
    print(s.cost())
```

```
<types.Stock object at 0x0000017BC2A4F9C8>
4555.0
```

"새로운 클래스 객체를 인스턴스화할 때 types.new_class() 함수를 사용할 수 있다. 간단히 클래스 이름과 키워드 매개변수, 클래스 딕셔너리를 만들 콜백과 멤버만 주면 나머지는 자동으로 처리된다."

클래스를 프로그램적으로 정의

```
import operator
import types
import sys

def named_tuple(classname, fieldnames):
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, (args))

    cls_dict['__new__'] = __new__

    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

"collections.nametuple() 함수와 같은 기능을 하는 간단한 구현."

```
if __name__ == '__main__':
    Point = named_tuple('Point', ['x', 'y'])
    print(Point)
    p = Point(4, 5)
    print(len(p))
    print(p.x, p[0])
    print(p.y, p[1])
    try:
        p.x = 2
    except AttributeError as e:
        print(e)
    print('%s %s' % p)
```

```
<class '__main__.Point'>
2
4 4
5 5
can't set attribute
4 5
```

정의 시 클래스 멤버 초기화


```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields_):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields_ = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields_):
            raise ValueError('{} arguments required'.format(len(cls._fields_)))
        return super().__new__(cls,args)

class Stock(StructTuple):
    _fields_ = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields_ = ['x', 'y']
```

정의 시 클래스 멤버 초기화

```
if __name__ == '__main__':
    s = Stock('ACME', 50, 91.1)
    print(s)
    print(s[0])
    print(s.name)
    print(s.shares * s.price)
    try:
        s.shares = 23
    except AttributeError as e:
        print(e)
```

```
('ACME', 50, 91.1)
ACME
ACME
4555.0
can't set attribute
```

"클래스 인스턴스를 생성할 때가 아닌, 정의할 때 클래스 일부를 초기화 하고 싶다."

"메타클래스는 클래스를 정의하는 시점에 실행되므로, 추가적인 작업을 하고 싶을 때 사용하면 된다."

3. 메타프로그래밍

3.1 메타프로그래밍_I

3.2 메타프로그래밍_II

3.3 메타프로그래밍_III

3.4 메타프로그래밍_IV

함수 주석으로 다중 디스패치 구현

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2,3)
s.bar('hello')
```

```
Bar 2: 2 3
Bar 2: hello 0
```

"파이썬은 인자에 주석을 붙일 수 있도록 허용하고 있으니, 주석과 동일안 인자의 타입을 check하는 메타 클래스를 구현하자."

함수 주석으로 다중 디스패치 구현

```
import inspect
import types

class MultiMethod:
    '''
    Represents a single multimethod.
    '''
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        '''
        Register a new method as a multimethod
        '''
        sig = inspect.signature(meth)

        types = []
```

함수 주석으로 다중 디스패치 구현


```
    for name, parm in sig.parameters.items():
        if name == 'self':
            continue
        if parm.annotation is inspect.Parameter.empty:
            raise TypeError(
                'Argument {} must be annotated with a type'.format(name)
                )
        if not isinstance(parm.annotation, type):
            raise TypeError(
                'Argument {} annotation must be a type'.format(name)
                )
        if parm.default is not inspect.Parameter.empty:
            self._methods[tuple(types)] = meth
        types.append(parm.annotation)

    self._methods[tuple(types)] = meth
def __call__(self, *args):
    '''
    Call a method based on type signature of the arguments
    '''
    types = tuple(type(arg) for arg in args[1:])
    meth = self._methods.get(types, None)
    if meth:
        return meth(*args)
    else:
        raise TypeError('No matching method for types {}'.format(types))
```

함수 주석으로 다중 디스패치 구현


```
 def __get__(self, instance, cls):
        '''
        Descriptor method needed to make calls work in a class
        '''
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class MultiDict(dict):
    '''
    Special dictionary to build multimethods in a metaclass
    '''
    def __setitem__(self, key, value):
        if key in self:
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)
```

함수 주석으로 다중 디스패치 구현


```
class MultipleMeta(type):
    '''
    Metaclass that allows multiple dispatch of methods
    '''
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()

class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)
import time
class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

함수 주석으로 다중 디스패치 구현

```
if __name__ == '__main__':
    s = Spam()
    s.bar(2, 3)
    s.bar('hello')
    s.bar('hello', 5)
    try:
        s.bar(2, 'hello')
    except TypeError as e:
        print(e)

    d = Date(2020, 6, 1)
    print(d.year, d.month, d.day)

    e = Date()
    print(e.year, e.month, e.day)
```

```
Bar 1: 2 3
Bar 2: hello 0
Bar 2: hello 5
No matching method for types (<class 'int'>, <class 'str'>)
2020 6 1
2020 5 30
```

함수 주석으로 다중 디스패치 구현


```
import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)
```

"메타클래스와 주석을 사용하지 않고, 데코레이터로 비슷한 구현을 할 수 있다."

함수 주석으로 다중 디스패치 구현

```
 def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
        print('Bar 2:', s, n)
```

함수 주석으로 다중 디스패치 구현

```
if __name__ == '__main__':
    s = Spam()
    s.bar(2, 3)
    s.bar('hello')
    s.bar('hello', 5)
    try:
        s.bar(2, 'hello')
    except TypeError as e:
        print(e)
```

```
Bar 1: 2 3
Bar 2: hello 0
Bar 2: hello 5
No matching method for bar
```

프로퍼티 메소드 중복 피하기

```
def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)
    return prop
```

"타입 확인과 같은 작업을 수행하는 프로퍼티를 반복적으로 정의하는 클래스를 작성 중이다. 이 코드를 단순화하고 중복을 피하고 싶다."

프로퍼티 메소드 중복 피하기

```
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)
    def __init__(self, name, age):
        self.name = name
        self.age = age

if __name__ == '__main__':
    p = Person('Dave', 39)
    p.name = 'Guido'
    try:
        p.age = 'Old'
    except TypeError as e:
        print(e)
```

```
age must be a <class 'int'>
```

"속성을 프로퍼티 메소드로 감싸는 단순한 클래스로 구현한다."

손쉬운 콘텍스트 매니저 정의

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{}: {}'.format(label, end - start))

with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

```
counting: 0.8227663040161133
```

"가장 간단히 콘텍스트 매니저를 작성하려면 contextlib 모듈의 @contextmanager 데코레이터를 사용한다."

손쉬운 콘텍스트 매니저 정의


```
from contextlib import contextmanager
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working

if __name__ == '__main__':
    items = [1, 2, 3]
    with list_transaction(items) as working:
        working.append(4)
        working.append(5)
    print(items)
    try:
        with list_transaction(items) as working:
            working.append(6)
            working.append(7)
            raise RuntimeError('oops')
    except RuntimeError as e:
        print(e)

    print(items)
```

손쉬운 콘텍스트 매니저 정의

```
[1, 2, 3, 4, 5]
oops
[1, 2, 3, 4, 5]
```

"예외가 발생하지 않은 경우에만 리스트를 수정한 사항이 반영된다는 점에 착안한 코드이다."

지역변수 문제 없이 코드 실행

```
a = 13
exec('b = a + 1')
print(b)
```

```
14
```

```
def test():
    a = 13
    exec('c = a + 1')
    print(c)

test()
```

```
NameError: name 'c' is not defined
```

"보이는 것처럼 마치 exec() 구문을 실행하지 않은 것처럼 NameError 예외가 발생했다. exec()의 결과를 추후 계산에 사용하고자 하는 경우 문제가 될 수 있다."

지역변수 문제 없이 코드 실행

```
def test():
    a = 13
    loc = locals()
    exec('c = a + 1')
    c = loc['c']
    print(c)

test()
```

```
14
```

"이 문제를 고치기 위해서는 exec()를 실행하기 전에 locals() 함수로 지역변수를 얻어 내야 한다. 이 함수를 실행한 직후에는 지역 딕셔너리에서 수정한 값을 얻을 수 있다."

지역변수 문제 없이 코드 실행


```
def test():
    a = 13
    loc = locals()
    exec('b = a + 1')
    b = loc['b']
    print(b)

def test1():
    x = 0
    exec('x += 1')
    print(x)

def test2():
    x = 0
    loc = locals()
    print('before:', loc)
    exec('x += 1')
    print('after:', loc)
    print('x =', x)
```

지역변수 문제 없이 코드 실행

```
def test3():
    x = 0
    loc = locals()
    print(loc)
    exec('x += 1')
    print(loc)
    locals()
    print(loc)

def test4():
    a = 13
    loc = { 'a' : a }
    glb = { }
    exec('b = a + 1', glb, loc)
    b = loc['b']
    print(b)
```

지역변수 문제 없이 코드 실행

```
if __name__ == '__main__':
    print(':::: Running test()')
    test()

    print(':::: Running test1()')
    test1()

    print(':::: Running test2()')
    test2()

    print(':::: Running test3()')
    test3()

    print(':::: Running test4()')
    test4()
```

지역변수 문제 없이 코드 실행

```
:::: Running test()
14
:::: Running test1()
0
:::: Running test2()
before: {'x': 0}
after: {'x': 1, 'loc': {...}}
x = 0
:::: Running test3()
{'x': 0}
{'x': 1, 'loc': {...}}
{'x': 0, 'loc': {...}}
:::: Running test4()
14
```

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()
    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)
```

"파이썬 소스 코드를 파싱하고 분석하는 프로그램을 만들고 싶다."

```
if __name__ == '__main__':
    code = '''
for i in range(10):
    print(i)
del i
'''
    top = ast.parse(code, mode='exec')

    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

"ast 모듈로 파이썬 소스 코드를 추상 신택스 트리로 컴파일하고 분석할 수 있다."

```
Loaded: {'print', 'i', 'range'}
Stored: {'i'}
Deleted: {'i'}
```

파이썬 바이트 코드 디스어셈블

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)
```

"파이썬 코드를 인터프리터가 사용하는 하위 레벨 바이트 코드로 디스어셈블 해서 내부 동작성을 분석하고 싶다."

파이썬 바이트 코드 디스어셈블

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
    print('Blastoff!')

for op, oparg in generate_opcodes(countdown.__code__.co_code):
    print(op, opcode.opname[op], oparg)
```

```
120 SETUP_LOOP 31774
0 <0> None
100 LOAD_CONST 27393
4 DUP_TOP None
114 POP_JUMP_IF_FALSE 29726
0 <0> None
100 LOAD_CONST 31746
0 <0> None
...
```

"파이썬 함수를 디스어셈블하는 데 dis 모듈을 사용한다."